Avaliação de sistemas NoSQL para o gerenciamento de dados em *workflows* científicos

Trabalho de Formatura Supervisionado
(MAC0499)

Gabriel Ferreira Guilhoto
Orientadora: Profa. Dra. Kelly Rosa Braghetto

Bacharelado em Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo

19 de dezembro de 2015

Sumário

1 Introdução

A computação tem ganhado relevância em vários domínios científicos por permitir a análise de grandes quantidades de dados, inviável manualmente. O processamento desses dados pode demandar ambientes distribuídos, e nesse caso é conveniente dispor de sistemas gerenciadores de *workflows* científicos (SGWCs) para coordenar os processos computacionais envolvidos. Um *workflow* científico é uma descrição de um experimento computacional de análise de dados em relação às atividades que o compõem e às dependências de dados entre elas. Define-se quais são as tarefas a serem executadas e uma ordem parcial, de acordo com os dados consumidos e produzidos por cada atividade [BC14].

Para lidar com os dados produzidos durante a execução de um *workflow* científico, geralmente se adota uma das seguintes estratégias: o uso de um sistema de arquivos distribuído, a utilização de um sistema de armazenamento baseado em objetos ou a eliminação dos dados intermediários. Num sistema de arquivos distribuído, os dados são armazenados em recursos computacionais remotos mas são acessados como arquivos locais. Um problema desse tipo de sistema é o fato de exigir modificações no sistema operacional, o que pode não ser possível quando se usa um serviço de computação em nuvem, por exemplo [JDB+12]. Um sistema de armazenamento baseado em objetos possibilita que os dados sejam acessados por múltiplas instâncias, mas a recuperação de um arquivo só pode ser feita por meio de uma única chave associada, que geralmente é seu nome [JDV+10]. Num *workflow*, pode ser necessário buscar objetos por atributos diferentes da chave, e assim essa solução não é suficiente. A terceira abordagem, de remoção dos dados intermediários, é pertinente quando o espaço para armazenar os dados é limitado [RSZ+07], mas prejudica a validação e a reprodutibilidade do experimento. Além desses problemas, esses métodos também não são adequados quando se deseja compartilhar os dados com outras instituições de pesquisa [Wat15].

Com o propósito de contornar as deficiências dessas soluções, é possível usar um sistema NoSQL para gerenciar os dados de um *workflow* científico. Um sistema NoSQL geralmente tem mecanismos de replicação e particionamento dos dados para a escalabilidade da aplicação, permite indexações em campos diferentes da chave e pode servir como um repositório compartilhado acessível por terceiros.

Neste trabalho, foram estudados sistemas NoSQL de diferentes tipos e foi feito um estudo de caso sobre o uso de alguns desses sistemas para o gerenciamento de dados num *workflow* em particular, com operações de manipulação

de dados similares às frequentemente encontradas em aplicações científicas. O *workflow* foi implementado com a ideia de abstrair as operações de inserção e recuperação de dados, de modo que o código das atividades mude pouco ao alterar o sistema NoSQL e o acesso aos dados fique encapsulado num módulo separado. A avaliação foi feita com dois sistemas NoSQL diferentes, o MongoDB e o Cassandra, usando máquinas da Nuvem USP.

1.1 Organização do texto

No capítulo 2, são caracterizados os sistemas NoSQL, focando nas motivações para seu surgimento e nas particularidades dos diferentes tipos. No capítulo 3, é feita uma descrição dos *workflows* científicos. No capítulo 4, explicam-se os detalhes dos experimentos realizados, em relação ao *workflow* usado e sua implementação, aos dados manipulados e ao ambiente de execução.

2 Sistemas NoSQL

Os sistemas gerenciadores de bancos de dados relacionais (SGBDRs) se consolidaram como escolha comum para a persistência de dados em apliações computacionais. Em relação a um sistema de arquivos, têm vantagens como a criação de índices, o controle de concorrência, a tolerância a falhas, restrições de integridade e a utilização de uma linguagem padronizada de manipulação e consulta (SQL).

Entretanto, pode existir uma diferença entre o modo como os dados são armazenados num banco de dados relacional e as estruturas de dados manipuladas por aplicações e mantidas na memória principal. No modelo relacional, os valores de uma tupla são simples (não podem ser listas ou registros aninhados), diferentemente de estruturas em linguagens orientadas a objetos, por exemplo. Portanto, para que uma aplicação possa persistir os seus dados em um SGBDR, ela precisa antes convertê-los para um formato compatível. Esse problema é conhecido como incompatibilidade de impedância (*impedance mismatch*).

Outra deficiência dos SGBDRs é relacionada à dificuldade de escalarem horizontalmente. Quando há necessidade de armazenar grandes conjuntos de dados e lidar com muitos usuários os acessando, é preciso aumentar os recursos computacionais. Isso pode ser feito de forma vertical (usando uma máquina mais potente) ou horizontal (com múltiplas máquinas mais simples). No entanto, SGBDRs geralmente não são projetados para funcionarem em aglomerados (*clusters*) com várias máquinas, e escalar verticalmente pode ser inviável devido ao custo alto e a limitações físicas.

Por causa desses inconvenientes dos SGBDRs, surgiram alternativas para o armazenamento de dados. Em meados da década de 2000, a Google e a Amazon publicaram artigos influentes sobre seus sistemas de armazenamento distribuídos (BigTable [CDG+08] e Dynamo [DHJ+07] respectivamente), usados internamente para lidar com os grandes volumes de dados pelos quais as empresas eram responsáveis.

O termo NoSQL apareceu em 2009 para se referir a sistemas parecidos com o BigTable e o Dynamo. Apesar de não haver um significado bem definido para o termo NoSQL, os sistemas classificados como NoSQL geralmente têm as seguintes características (ou pelo menos boa parte delas): [FS13, cap. 1]

- São projetados para serem executados em aglomerados com várias máquinas e não garantem a mesma consistência de dados das transações ACID dos SGBDRs.

- Não usam o modelo de dados relacional.
- Não têm interfaces ou linguagens padronizadas para a recuperação e manipulação dos dados como a SQL.
- Lidam com bancos de dados sem esquema definido, de forma que atributos possam ser adicionados nos registros sem a necessidade de mudar a estrutura dos bancos antes.
- São projetos de código aberto.

2.1 Teorema CAP

Para poderem funcionar de forma distribuída, os sistemas NoSQL não conseguem assegurar bom desempenho e manter as mesmas propriedades que um SGBDR garante. O motivo para isso acontecer é explicado pelo teorema CAP (de ***c****onsistency*, ***a****vailability* e ***p****artition tolerance*), que foi apresentado como uma conjectura por Eric Brewer em 2000 [Bre00] e provado formalmente por Seth Gilbert e Nancy Lynch em 2002 [GL02]. Segundo o teorema, num sistema distribuído não é possível obter por completo as três seguintes propriedades: consistência, disponibilidade e tolerância a partições. Gilbert e Lynch formalizaram essas três características da seguinte forma: [GL02] [FS13, cap. 5]

- A consistência ocorre quando existe uma ordem total em todas as operações, de modo que cada uma delas possa ser vista como se fosse feita num único instante, isoladamente das demais. Como consequência, uma leitura deve devolver o último valor escrito.
- Para a disponibilidade ser satisfeita, toda solicitação recebida por um nó que não esteja em falha deve resultar numa resposta. No caso de um sistema NoSQL, se é possível conversar com um nó também deve ser possível ler e gravar dados nele.
- A tolerância a partições significa que a rede permite perdas de mensagens de um nó a outro. Ou seja, o *cluster* deve continuar a funcionar mesmo com falhas na comunicação que o dividam em partições incomunicáveis.

A demonstração do teorema justifica que não é possível satisfazer essas três condições, mas é factível escolher duas delas, nas seguintes situações:

1. Consistência e disponibilidade: faz sentido quando não há a possibilidade de partição da rede. Por exemplo, num sistema com um único servidor,

que não pode ser particionado, é plausível fornecer consistência e disponibilidade. É o caso da maioria dos SGBDRs. Entretanto, para se criar um *cluster* consistente e disponível, seria necessário que todos os nós saíssem do ar quando fosse acontecer uma partição. Dessa forma, a disponibilidade como definida pelo teorema não seria violada, pois não haveria nós sem falhas. Na prática, isso é inviável pois seria preciso garantir que uma partição ocorra com pouca frequência, e conseguir detectá-la em tempo hábil não é trivial [FS13, cap. 5].

2. Consistência e tolerância a partições: se não é preciso disponibilidade, as outras duas condições podem ser satisfeitas trivialmente com um sistema que ignora todas as solicitações. Outra possibilidade é a de existir um nó mestre que mantém o valor de um objeto. Se outro nó receber uma solicitação, ele a encaminha para o mestre, cuja resposta é depois conduzida ao cliente [GL02]. A consistência é satisfeita já que todas as solicitações passam pelo mestre. A disponibilidade é sacrificada pois, se houver uma partição que separa um nó escravo do mestre, o escravo não consegue responder às requisições.

3. Disponibilidade e tolerância a partições: se a consistência não é necessária, uma solução simples é a de devolver o valor inicial de um objeto como resposta a todas as solicitações [GL02].

Num *cluster*, a tolerância a partições é importante pois a opção 1 é impraticável. Portanto não é possível garantir consistência e disponibilidade completamente, e um sistema NoSQL precisa fazer escolhas em relação a essas propriedades. Embora o teorema aparente indicar que exista uma decisão binária entre consistência e disponibilidade, é possível obter ambas as características parcialmente. Por exemplo, num sistema de reserva de hotéis com dois nós, um deles pode ser designado como mestre e processar solicitações de escrita e leitura, ao passo que o outro, o escravo, trataria diretamente apenas as leituras. Assim, se houvesse uma falha de comunicação, os usuários conectados ao nó escravo seriam capazes de ver informações possivelmente obsoletas sobre as reservas, mas não poderiam criar uma nova reserva e não gerariam insconsistências nos dados gravados. Dessa forma, não há consistência e disponibilidade completas como nas definições do teorema, mas um pouco de cada. Outra opção é permitir escritas nos dois nós para garantir a disponibilidade completa, o que relaxa a consistência pois poderia haver mais reservas do que quartos livres. É possível, contudo, que o hotel prefira lidar com esse problema em vez de perder uma reserva por falhas na rede. Ou seja, a decisão sobre como balancear a consistência

e a disponibilidade não é universal e depende do domínio da aplicação [FS13, cap. 5].

2.2 Replicação e particionamento

Para poderem ser executados eficientemente em aglomerados com várias máquinas, os sistemas NoSQL recorrem a pelo menos uma de duas estratégias: replicação e particionamento de dados. A replicação consiste em copiar os dados de um nó para outro(s), e o particionamento (ou fragmentação) significa separar o conjunto de dados em nós diferentes.

O particionamento pode melhorar o desempenho pois permite que usuários diferentes acessem partes distintas do conjunto de dados independentemente, de forma paralela. Entretanto, não garante maior tolerância a falhas pois um problema em um nó torna inacessíveis todos os dados dele.

A replicação pode ser feita de duas formas: com a abordagem mestre-escravo ou ponto a ponto. Na distribuição mestre-escravo, existe um nó mestre, responsável por processar a escrita de dados, e outros nós escravos, capazes apenas de lidar com leituras e que devem se sincronizar com o mestre. A adição de nós ao sistema aumenta o desempenho da leitura, mas não melhora o da gravação. Se um dos nós falhar, mesmo que seja o mestre, as leituras ainda podem ser feitas nos outros nós. Em caso de falha do mestre, ainda é possível que um escravo seja designado como mestre e passe a lidar com as escritas. Contudo, um problema intrínseco à replicação é a inconsistência. Como existe uma demora para que uma atualização se propague para todos os nós escravos, é possível que clientes acessando nós distintos vejam dados diferentes.

Na replicação ponto a ponto (*peer-to-peer*), não existe um mestre. Todos os nós podem tratar gravações e é possível escrever mesmo havendo falha em um nó qualquer. Mas nesse caso a consistência é ainda mais sacrificada, pois podem acontecer duas gravações em nós diferentes ao mesmo tempo. Para minimizar esse inconveniente, o sitema NoSQL pode definir um limite mínimo de propagação da escrita entre os nós para que a operação seja bem-sucedida, ou então tentar fundir as alterações conflitantes (como no Git).

A replicação e o particionamento podem ainda ser usados conjuntamente. Se a replicação for mestre-escravo, a categorização de um nó como mestre ou escravo é relativa ao conjunto de dados. Um nó pode ser mestre para uma partição e escravo para outra, por exemplo [FS13, cap. 4].

2.3 Tipos de sistemas NoSQL

Neste trabalho, serão estudados três tipos de sistemas NoSQL: documentos, famílias de colunas e chave-valor.

2.3.1 Documentos

Um banco de dados de documentos é composto de documentos em formatos como XML, JSON e BSON, que são comparáveis a linhas (ou tuplas) num SGBDR. Por exemplo, um documento correspondente a um cliente de uma loja de comércio eletrônico pode ser:

```
{
    "id": 1,
    "nome": "João",
    "endereço":
    {
        "rua": "xxx",
        "número": 21,
        "cidade": "São Paulo"
    },
    "telefone": ["3333-3333", "99999-9999"]
}
```

É possível aninhar documentos dentro de outros (como acontece no atributo `"endereço"`), e um atributo pode ter uma lista de valores, como no campo `"telefone"`.

E um documento correspondente a outro cliente poderia ser:

```
{
    "id": 2,
    "nome": "Maria",
    "e-mail": "maria@maria.com",
    "telefone": "4444-4444"
}
```

Ou seja, a princípio não é preciso que todos os documentos de uma coleção tenham a mesma estrutura. E não há atributos definidos com "NULL" como haveria em um SGBDR; simplesmente não se coloca o atributo no documento se não houver a informação. É possível também criar um documento com um

atributo novo (que não aparece em nenhum dos documentos já existentes) sem dificuldade [FS13, cap. 9].

O MongoDB e o CouchDB são dois exemplos de sistemas NoSQL de documentos bastante usados. As principais características desses sistemas são apresentadas a seguir.

MongoDB Uma coleção no MongoDB é similar a uma tabela num SGBDR, pois agrupa documentos relacionados (que no entanto não precisam ter exatamente a mesma estrutura).

O MongoDB permite configurar replicação similar à mestre-escravo para se prevenir de falhas em algum servidor e/ou aumentar a capacidade de leitura. Define-se um conjunto de réplicas (*replica set*) e escolhe-se uma delas como instância primária (o mestre). Operações de escrita só podem ser recebidas pelo mestre, que registra todas as alterações nos dados num registro de operações (*oplog*). As demais instâncias são chamadas de secundárias e podem receber apenas solicitações de leitura (são os escravos). Quando há uma operação de escrita, as instâncias secundárias aplicam as operações registradas no *oplog* do mestre de maneira assíncrona [Monb].

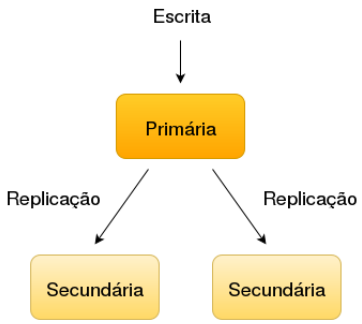

Figura 1: Replicação de uma operação de escrita no MongoDB

Por padrão, as leituras são sempre direcionadas à instância primária. Desse modo, a finalidade da replicação é apenas de permitir a disponibilidade em caso de falhas na instância primária, e uma leitura é sempre feita sobre a versão mais recente de um documento. Contudo, é possível permitir leituras nas instâncias secundárias para melhorar o desempenho caso eventuais resultados desatualizados não sejam um problema [Mona].

Quando não há comunicação da instância primária com as demais por mais de dez segundos, escolhe-se uma das secundárias para assumir o papel de primária por meio de uma eleição [Monb].

Também é possível dividir dados em múltiplas máquinas com o MongoDB. O particionamento separa o conjunto de dados em vários servidores, que são chamados de partições (*shards*). O objetivo é reduzir o número de operações com que cada máquina tem de lidar e por consequência aumentar o desempenho de leitura e gravação. Cada partição pode ainda ser um conjunto de réplicas para aumentar a disponibilidade. Além dos servidores com as partições, é preciso haver um ou mais *query routers*, que se conectam com o cliente e direcionam as operações para a partição adequada. Também são necessários três servidores de configuração (*config servers*) que armazenam os metadados de mapeamento usados pelos *query routers* para determinarem a partição de um documento.

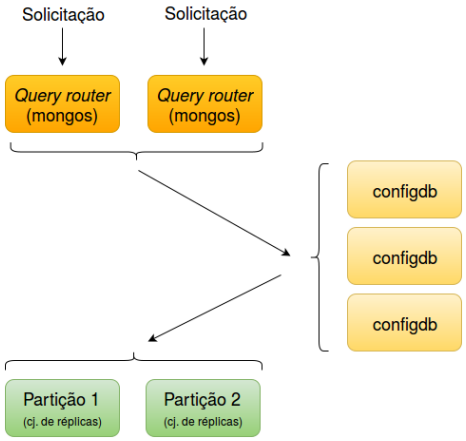

Figura 2: Particionamento no MongoDB

O particionamento é feito com base em uma chave de partição (*shard key*), que é um campo ou conjunto de campos indexado que precisa estar presente em todos os documentos da coleção. Os documentos podem ser distribuídos com base em intervalos, de modo que cada partição tenha apenas documentos cujo valor da chave de partição esteja num intervalo pré-definido. Assim documentos com chaves de partição próximas ficam agrupadas num mesmo servidor. Também é possível distribuí-los com base na aplicação de uma função de *hashing* sobre a chave de partição, o que separa melhor documentos com chaves parecidas [Monc].

CouchDB O CouchDB é um sistema de gerenciamento de bancos de dados de documento mantido pela Apache que usa documentos JSON e recebe requisições por HTTP [Apaa]. O sistema foca na replicação de dados para a escalabilidade horizontal, valorizando ganhos de desempenho em detrimento da consistência

constante. É permitido realizar escritas em qualquer nó sem ter de esperar a sincronização com outros servidores [Apab].

Cada documento no CouchDB está associado a um número de versão. Em caso de modificação em um documento, é criada uma nova versão com outro número e mantida junto com a antiga. Se há duas modificações discrepantes num documento em dois servidores diferentes, ambas as versões são marcadas como em conflito e mantidas nos dois servidores após a replicação. Caso se tente ler esse documento em qualquer um dos dois servidores, por padrão é usado um algoritmo determinístico (que sempre escolhe uma versão em específico) para se decidir qual devolver, mas é possível solicitar informações sobre conflitos. É responsabilidade da aplicação verificar a existência de versões divergentes e tratar os conflitos adequadamente. Mas esse comportamento acontece apenas em casos de conflito entre servidores; caso haja apenas um nó e se tente salvar uma modificação sobre um documento lido antes de outra alteração, é devolvido um erro [Apab, Apac].

Apesar dos recursos relacionados à replicação, não existem funcionalidades para particionamento de dados na versão estável mais recente do CouchDB durante a realização do trabalho (1.6). Existe uma versão *2.0 Developer Preview* em desenvolvimento com suporte ao particionamento, mas a documentação é pouca e não foi possível explorar o funcionamento adequadamente.

2.3.2 Famílias de colunas

Um banco de dados de família de colunas armazena dados como linhas com várias colunas associadas.

Cassandra No Cassandra, a unidadade básica de armazenamento é a coluna, que é um par chave-valor junto com um *timestamp*. Uma linha é um conjunto de colunas associadas à mesma chave, e uma família de colunas é uma coleção de linhas semelhantes (análogo a uma tabela num SGBDR, com a diferença de que as linhas não precisam ter as mesmas colunas). Um *keyspace* é um conjunto de família de colunas relacionadas a uma aplicação, como um esquema num SGBDR. Uma coluna pode ainda ser composta de outras colunas, e nesse caso é chamada de supercoluna [FS13, cap. 10].

O Cassandra implementa funcionalidades de replicação e particionamento. A estratégia padrão para o particionamento usa a aplicação da função `MurmurHash`[1] ao valor da chave da linha para determinar o nó responsável por ela, com o ob-

[1] Antes era utilizada por padrão uma função de *hashing* MD5, mas ela foi substituída pela `MurmurHash`, outra função com melhor desempenho [Data].

jetivo de distribuir os dados uniformemente entre os nós. Outra possibilidade é de separar por ordem da chave, o que pode desbalancear a quantidade de dados em cada nó.

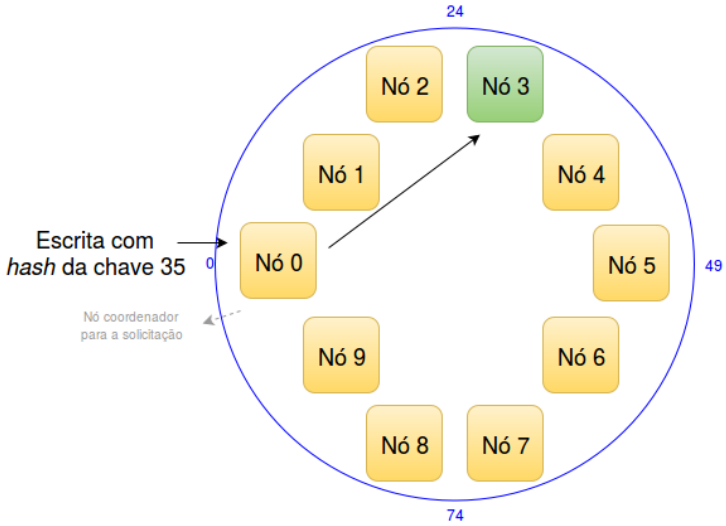

Figura 3: Exemplo de particionamento no Cassandra num caso fictício em que os valores da função de *hashing* vão de 0 a 99. O nó 0, que recebe a operação de escrita, é capaz de calcular o *hash* da chave e determinar que a linha deve ser escrita no nó 3.

Para replicar, é preciso definir um fator de replicação para o *keyspace*, que determina a quantidade de nós em que uma linha estará. Para se decidir quais serão esses nós, existem duas estratégias: a simples, que escolhe os próximos nós em sentido horário a partir do nó escolhido pelo particionador, e a de topologia de rede, que dá mais flexibilidade para a designação dos nós [Datc].

A replicação no Cassandra é ponto a ponto; ou seja, é possível ler e escrever a partir de qualquer nó, e o sistema cuida da replicação e do particionamento para os nós correspondentes. Quando um nó recebe uma solicitação de escrita, ela é primeiramente registrada num *commit log* em disco, que permite a recuperação em caso de falhas, e depois numa estrutura em memória chamada *memtable*, que é transferida para a *SSTable* em disco quando está cheia. As escritas são atômicas por linha, mas não há transações como em SGBDRs. O Cassandra adota a ideia da *tunable data consistency*, que permite definir a consistência desejada a cada operação. Ao se fazer uma escrita, pode-se escolher uma estratégia dentre várias para a consistência, como esperar a escrita em um nó qualquer, em um dos nós responsáveis pela linha, na maioria dos nós responsáveis pela linha ou

em todos os responsáveis. Existem opções semelhantes para uma leitura, que pode exigir por exemplo a resposta de uma réplica, da maioria delas ou de todas [Datb].

2.3.3 Chave-valor

Um banco de dados chave-valor funciona basicamente como uma tabela de *hashing* persistente, associando uma chave a um valor. O valor pode ter qualquer formato, que é responsabilidade da aplicação e ignorado pelo sistema . Por isso o acesso pode ser feito apenas pela chave [FS13, cap. 8].

Riak No Riak (também conhecido como Riak KV), os pares chave-valor são agrupados em *buckets*, de modo que um *bucket* não possa ter chaves repetidas.

Num *cluster* Riak, cada servidor físico (nó) executa um número de nós virtuais, que são chamados de *vnodes*. Cada *vnode* é responsável por uma partição do conjunto de pares *bucket*-chave, e cada nó corresponde a um número de *vnodes* aproximadamente igual.

Não existe um nó mestre e todos os nós são capazes de processar qualquer solicitação. Numa inserção que pede que o número de cópias seja N, descobre-se o *vnode* responsável pelo dado e escreve-se nele e em $N-1$ outros *vnodes* [Bas].

3 *Workflows* científicos

A computação tem ganhado um papel relevante na pesquisa científica por permitir simulações de fenômenos complexos inviáveis manualmente. Grandes volumes de dados experimentais podem ser processados por sistemas de *software* complexos para se gerar novos dados que então são analisados pelos cientistas. Para se descrever todas as etapas de um experimento científico computacional e as dependências entre elas, pode-se usar modelos de "fluxos de trabalho", mais conhecidos como *workflows*.

Um modelo de *workflow* define as atividades a serem realizadas para alcançar um objetivo científico e determina a ordem na qual elas podem ser executadas. Uma atividade (também chamada de tarefa, processo, ação ou transição) é uma unidade atômica de trabalho no *workflow*, e as relações de precedência entre as atividades são indicadas por conectores. Uma execução particular de um dado *workflow* é chamada de instância.

Os *workflows* são usados desde a década de 1990 para automatizar processos industriais e gerenciais, e nesse caso são chamados de *workflows* de negócio. Houve muito trabalho para padronizar as linguagens de modelagem de *workflows* de negócio, como os modelos da Workflow Management Coalition (WfMC) e da Business Process Model and Notation (BPMN). *Workflows* passaram a ser usados em aplicações científicas apenas na década de 2000 e ainda em poucos domínios. Como consequência, não existem representações padrão para os modelos de *workflows* científicos, de modo que cada sistema de gerenciamento adota uma linguagem de modelagem própria.

Em *workflows* de negócio, existe uma preocupação importante na integridade da sequência de atividades, pois elas podem envolver transações comerciais. Em caso de falhas, é necessário poder desfazer ou compensar as atividades já realizadas. Já em *workflows* científicos, esse não é um problema comum. Se ocorrer alguma falha, é suficiente eliminar os resultados produzidos pela atividade correspondente e reiniciar a execução do *workflow* a partir do início ou do ponto de interrupção.

A modelagem de um *workflow* pode ser feita por diversas perspectivas, entre as quais estão a de fluxo de controle e a de fluxo de dados. Um fluxo de controle (exemplificado na figura 4) descreve a ordem de execução das atividades por construtores de composição, permitindo que uma atividade execute quando as suas antecessoras são concluídas. Um fluxo de dados (como na figura 5) indica a dependência entre os dados manipulados, de forma que atividades produzem dados que serão consumidos por outras atividades.

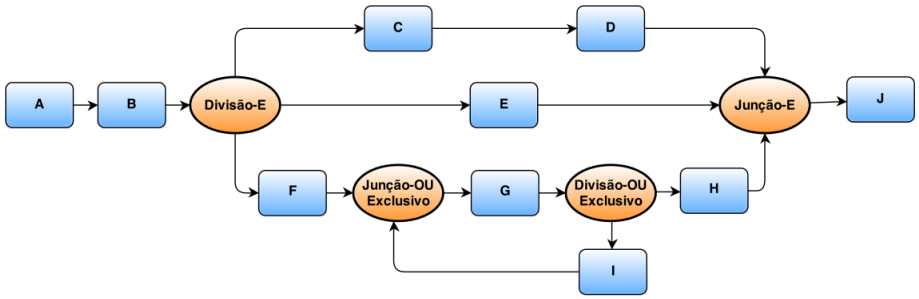

Figura 4: Exemplo de *workflow* modelado com base no fluxo de controle, obtido de [BC14]. A atividade B só pode ser iniciada assim que A terminar. Quando B for concluída, C, E e F podem ser executadas paralelamente. Apenas uma dentre H e I pode ser executada quando G finalizar. A junção-OU-exclusivo indica que G estará pronta para executar quando uma dentre F ou I tiver terminado. E a junção-E expressa que a atividade J deve aguardar o fim de D, E e H para iniciar.

Boa parte do trabalho realizado para a modelagem de *workflows* de negócio utiliza principalmente a perspectiva de fluxos de controle. Em contrapartida, os *workflows* científicos geralmente envolvem a manipulação intensiva de grandes quantidades de dados, pois o objetivo principal é a geração de dados derivados de dados brutos. Por isso eles são caracterizados como baseados em fluxos de dados (*data-driven*).

Na abordagem de fluxo de dados, as construções básicas são o processamento, o *pipeline*, a distribuição, a agregação e a redistribuição.

O processamento é a construção mais simples e consiste na manipulação de dados de entrada para produzir dados de saída, como feito pela atividade `mAdd` na figura 5. Um *pipeline* é uma série de processamentos em que uma atividade recebe como entrada os dados de saída da atividade anterior, como na sequência mAdd–mShrink–mJPEG do Montage. A distribuição de dados é feita por uma atividade que produz múltiplos conjuntos de dados usados por atividades distintas. Um caso particular de distribuição é chamado de particionamento e ocorre quando uma atividade que recebe um grande conjunto de dados divide-os em subconjuntos menores a serem processados por várias atividades, a exemplo da mBgModel. A agregação de dados acontece quando uma atividade processa dados de saída de mais de uma atividade, como faz a mConcatFit. E a redistribuição de dados é o caso de uma atividade que recebe vários conjuntos de dados e gera outros vários, combinando a agregação com a distribuição. [BCD+08, BC14]

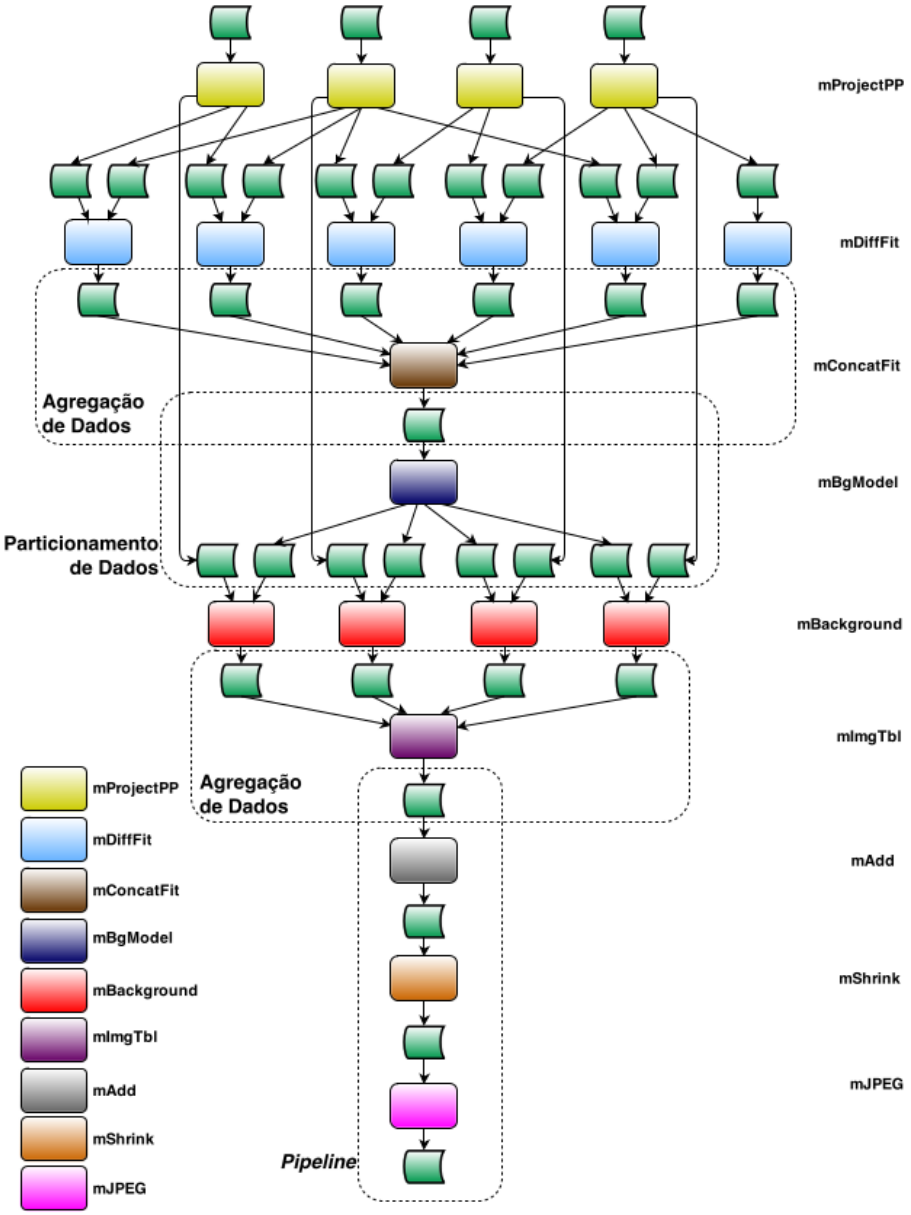

Figura 5: *Workflow* científico Montage [Cal], que exemplifica várias construções de fluxo de dados. Os retângulos arredondados são atividades e as formas verdes são dados de entrada/saída. Imagem obtida de [BC14].

3.1 Pegasus

Existem sistemas de gerenciamento de *workflows* científicos (SGWCs) que servem de apoio à definição, à execução e ao monitoramento de *workflows* científi-

ficos. Um SGWC é capaz de executar aplicações científicas compostas por várias atividades a partir de uma representação digital que define a ordem delas. Geralmente os SGWCs também fornecem ferramentas para permitir a reprodutibilidade dos experimentos, a execução em sistemas computacionais de alto desempenho, o monitoramento da execução e o tratamento de falhas.

O Pegasus [DSS+05] é um SGWC que permite a execução de *workflows* em recursos distribuídos heterogêneos. Ele recebe uma descrição do *workflow* na linguagem DAX (*Directed Acyclic Graph in XML*) e informações sobre os recursos disponíveis para então gerar um *workflow* executável. Na linguagem DAX, um *workflow* é um grafo de atividades, cada uma com um código de identificação e informações sobre arquivos de dados de entrada e saída, o que a caracteriza como orientada a fluxo de dados [BC14].

O componente Workflow Mapper do Pegasus é o responsável por gerar um *workflow* executável a partir do modelo fornecido pelo usuário, localizando o *software*, os dados e os recursos computacionais necessários. A execução do *workflow* é gerenciada pelo DAGMan, cujas responsabilidades incluem o gerenciamento de dados, o monitoramento do estado das atividades e o tratamento de erros. O gerenciamento das atividades individualmente é feito pelo escalonador de tarefas HTCondor, que supervisiona a execução nos recursos locais e remotos [OCFdS+15].

4 Experimentos

Foram feitos experimentos para comparar o desempenho do MongoDB com o do Cassandra para gerenciar os dados de um *workflow* científico para o Pegasus. Esse *workflow* faz análise de dados de rastros de um *cluster* do Google.

4.1 Dados usados

Foram usados dados do Google Cluster Data, um conjunto de dados em arquivos CSV com rastros dos sistemas de gerenciamento de *clusters* do Google. Os dados têm informações coletadas durante 29 dias a partir de 1º de maio de 2011, num *cluster* com cerca de 12 500 máquinas localizado no leste dos Estados Unidos. O *cluster* recebe trabalho na forma de *jobs*, compostos de uma ou mais tarefas (*tasks*). Uma tarefa é programa a ser executado em uma única máquina. Foi usada a tabela `task_events`, que tem informações sobre eventos (submissão, execução, falha etc.) de tarefas e tem os seguintes campos: [RWH11]

- `time`, um inteiro de 64 bits com o tempo de início da tarefa em microssegundos desde 600 segundos antes do início do período do rastreamento
- `job ID`, que identifica o *job* a que a tarefa pertence
- `task index`, o índice da tarefa em relação ao *job* a que pertence
- `machine ID`, o número da máquina na qual a tarefa foi executada
- `event type`, o tipo do evento, que pode ser:
  - 0 (`SUBMIT`), se a tarefa foi submetida e pode ser escalonada
  - 1 (`SCHEDULE`), se a tarefa foi escalonada numa máquina
  - 2 (`EVICT`), se a tarefa foi desescalonada por causa de ...
  - 3 (`FAIL`), se a tarefa foi desescalonada por falha na própria tarefa
  - 4 (`FINISH`), se a tarefa foi terminada normalmente
  - 5 (`KILL`), se a tarefa foi cancelada pelo usuário
  - 6 (`LOST`), se a tarefa supostamente terminou mas não existia registro de seu término
  - 7 (`UPDATE_PENDING`), se os dados foram alterados enquanto a tarefa estava esperando ser escalonada
  - 8 (`UPDATE_RUNNING`), se os dados foram alterados enquanto a tarefa estava escalonada

- `user`, nome do usuário responsável pela submissão da tarefa, passado por uma função de *hashing*
- `scheduling class`, inteiro de 0 a 3 que indica o quão sensível a latência a tarefa é. O valor 3 indica alta sensibilidade (por exemplo, a resposta a uma requisição de usuário). Tarefas com alta sensibilidade à latência tendem a ter alta prioridade.
- `priority`, inteiro não negativo que indica a prioridade. Quanto maior o valor, a tarefa tem mais preferência na obtenção de recursos.
- `CPU request`, consumo máximo de CPU permitido para a tarefa, em segundos de *cores* por segundo, normalizado para ser um valor de 0 a 1
- `memory request`, consumo máximo de memória permitido para a tarefa, em *bytes*, normalizado para ser um valor de 0 a 1
- `disk space request`, consumo máximo de disco permitido para a tarefa, em *bytes*, normalizado para ser um valor de 0 a 1
- `different machines restriction`, que é 1 se a tarefa precisa ser escalonada numa máquina diferente de todas as outras tarefas do mesmo *job* e 0 caso contrário

A chave da tabela é composta pelos campos `timestamp`, `job ID` e `task index`.

4.2 *Workflow* usado

Os experimentos foram feitos com base num *workflow* científico criado por Elaine Naomi Watanabe, mestranda do programa de pós-graduação do Departamento de Ciência da Computação do IME-USP. Seu trabalho tem como objetivo propor novas estratégias de gerenciamento de dados em *workflows* científicos executados em nuvens computacionais. O modelo de *workflow* foi definido para testar técnicas desenvolvidas no trabalho do mestrado e foi usado por conter operações de manipulação de dados frequentemente encontradas nos *workflows* científicos.

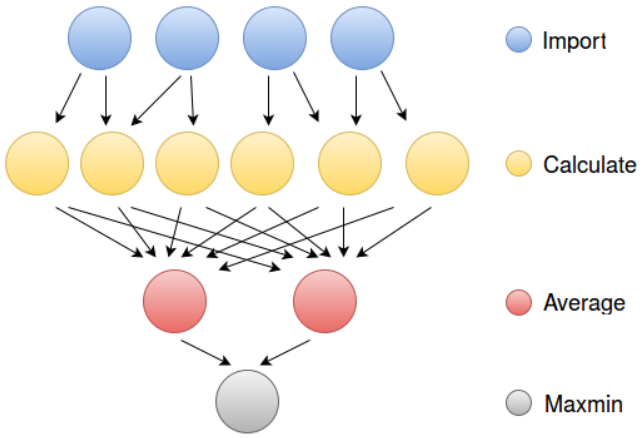

Figura 6: Representação simplificada do *workflow* usado nos experimentos. No *workflow* real as atividades são em maior número.

O *workflow* tem quatro tipos de atividade:

- Cada atividade `Import` recebe um arquivo CSV compactado e insere seus dados numa coleção `task_events` no sistema NoSQL. Essa coleção tem todos os atributos descritos na seção 4.1 e mais dois campos, `filepath` e `numline`, para indicar o arquivo de origem do registro e o número da linha em que estava, respectivamente.

- Cada atividade `Calculate` obtém um subconjunto dos dados da coleção `task_events`, calcula a razão entre os consumos de CPU e memória de cada objeto e armazena o resultado numa coleção `ratio` no sistema NoSQL. A coleção `ratio` tem os campos `time`, `job ID`, `bigint`, `task index`, `event type`, `numline` e `filepath` copiados da coleção `task_events`, além do campo `ratio cpu memory`, com a razão calculada.

- Cada atividade `Average` obtém os objetos da coleção `ratio` com um determinado `event type` e calcula a média das razões entre CPU e memória. O resultado é armazenado na coleção `average_ratio`, com os atributos `event type` (copiado da `ratio`), `total tasks` (com a contagem total de ocorrências do `event type`), `total valid tasks` (com o valor de `total tasks` subtraído do número de eventos sem dados de consumo de memória/CPU), `sum ratio cpu memory` (com a soma das razões), `mean ratio cpu memory` (com a média das razões).

- A atividade `Maxmin` lê todos os objetos da coleção `average_ratio`, determina os tipos de evento com maior e menor média de razão entre consumo

de CPU e memória e armazena um objeto na coleção `analysis_ratio` com os campos `ratio max` (com a média máxima), `event type max` (com o tipo de evento correspondente à média máxima), `ratio min` (com a média mínima) e `event type min` (com o tipo de evento correspondente à média mínima).

Há quatro atividades `Import` (uma para cada arquivo CSV), 32 atividades `Calculate` (uma para cada *core* disponível), nove atividades `Average` (uma para cada `event type`) e uma atividade `Maxmin`.

4.2.1 Implementação

As atividades do *workflow* são escritas em Python [Pyt]. Para acessarem o sistema NoSQL ou obterem os dados de um arquivo CSV compactado (no caso da atividade `Import`), criam objetos da classe `DataStoreClient`, cujo construtor tem dois parâmetros: `type` e `config`. O parâmetro `type` é uma *string* que pode ser `"mongodb"`, `"cassandra"` ou `"csv.gz"`, caso faça acesso ao MongoDB, ao Cassandra ou aos dados de um arquivo CSV compactado, respectivamente. O parâmetro `config` é um dicionário com valores que descrevem como deve ser feito o acesso aos dados, tais como nome do arquivo no caso do CSV e endereços das máquinas, nome da coleção e tipo de operação no caso dos sitemas NoSQL. Para armazenar ou recuperar os dados, são usados os seguintes métodos da classe `DataStoreClient`:

- `saveData(data)`: Recebe em `data` uma lista de objetos em forma de dicionários e os insere na coleção especificada na configuração. É usado por todas as atividades para armazenar os resultados no sistema NoSQL.
- `getData()`: No caso do CSV compactado, lê o arquivo e devolve uma lista com um dicionário para cada linha de dados, contendo os campos do arquivo acrescidos de `filepath` (com o nome do arquivo de origem) e `numline` (com o número da linha do registro). No caso dos sistemas NoSQL, o comportamento depende do tipo da operação definido no construtor, que pode ser:
  - `"UNIT"`, e nesse caso o dicionário de configuração deve informar o `numline` e o `filepath` do primeiro e do último item a serem processados. São recuperados os objetos no intervalo especificado na coleção, devolvidos numa lista de dicionários. É utilizado pelas atividades `Calculate`.

– `"GROUP_BY_COLUMN"`, e então são devolvidos os objetos da coleção que têm um atributo com um valor específico. O atributo e o valor são informados no dicionário de configuração. É utilizado pelas atividades `Average`.
– `"ALL"`, que devolve todos os objetos da coleção. É utilizado pela atividade `Maxmin`.

4.3 Ambiente

Para realizar os experimentos, foram utilizadas dez máquinas da Nuvem USP [ZKLH13], com as seguintes configurações:

Número da máquina	1	2	3	4	5	6	7	8	9	10
Memória RAM	32 GB									
Clock da CPU	2300 MHz									
Número de *cores*	8									
Sistema operacional	Ubuntu Server 12.04						Ubuntu 14.04			
Discos	20 GB				120 GB			220 GB		
Processos no experimento com o MongoDB	HTCondor				Pegasus e `mongod`		`mongod` e `mongos`	`mongod` e `configdb`		
Processos no experimento com o Cassandra	HTCondor				Pegasus e Cassandra		Cassandra			

Foram usadas as versões 3.0.6 do MongoDB, 2.1.11 do Cassandra e 4.3.2 do Pegasus.

No caso do MongoDB, as máquinas estão dispostas da seguinte forma:

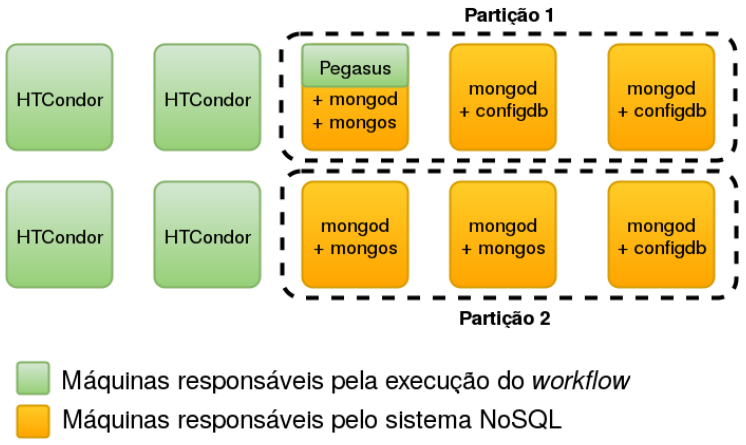

Figura 7: Disposição das máquinas no experimento com o MongoDB

As máquinas com o HTCondor são as responsáveis por executar o *workflow* propriamente dito, e o Pegasus monitora o estado das atividades para determinar qual pode executar em cada momento. Como não faz processamento pesado, a máquina em que o Pegasus está é compartilhada com o MongoDB. Os `mongod` armazenam os dados da aplicação, os `configdb` têm metadados sobre a partição em que cada documento se encontra e os `mongos` recebem as conexões de clientes. Como os `configdb`s demandariam armazenar mais dados, escolheu-se as máquinas com mais espaço em disco para eles. Há duas partições, cada uma com um conjunto de três réplicas, o que significa que os dados são divididos em duas partes e cada uma dessas partes está presente em três máquinas diferentes.

No caso do Cassandra, as máquinas estão dispostas da seguinte forma:

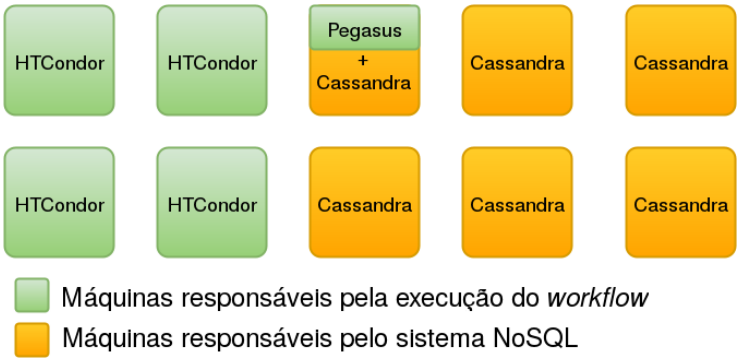

Figura 8: Disposição das máquinas no experimento com o Cassandra

A disposição no Cassandra é mais simples pois não existe a ideia de partições fixas e as máquinas têm todas a mesma responsabilidade. O *keyspace* utilizado foi configurado com um fator de replicação 3, ou seja, cada linha está presente em três máquinas.

Cada experimento consistiu em trinta repetições da execução do *workflow* seguida da remoção dos dados do sistema NoSQL. O início de cada execução foi feito aproximadamente uma hora depois do início da última execução. Foram usados os tempos medidos pelo Pegasus.

5 Resultados e conclusões

Para cada tipo de atividade, foi medido o consumo de tempo entre o início da primeira instância e o término da última. No gráfico abaixo, estão representadas as médias de tempo de execução com o intervalo de confiança de 95%.

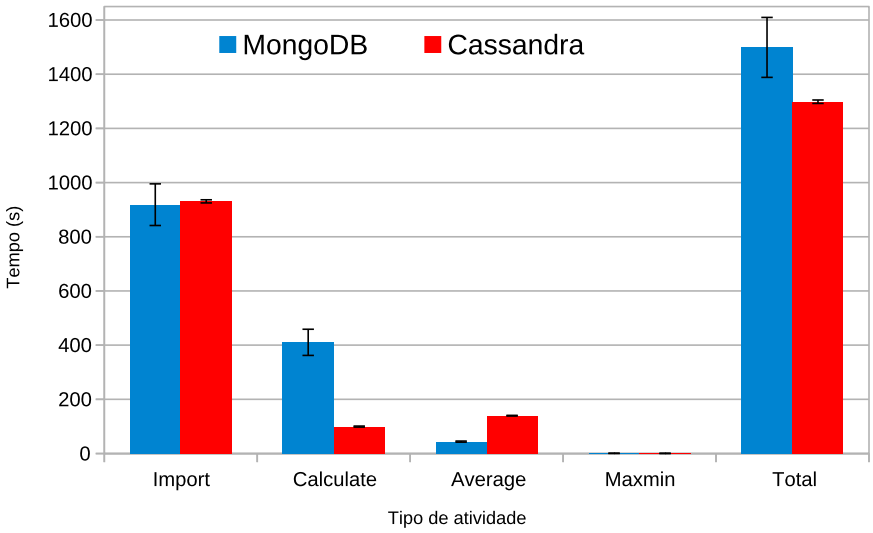

Figura 9: Tempos de execução do *workflow*

A tabela abaixo indica qual sistema teve desempenho melhor de acordo com o tipo de operação das atividades. As atividades `Maxmin` não foram consideradas pois o tempo de execução é insuficiente para se tirar conclusões.

Tipo de atividade	**Tipo de operação**	**Sistema com melhor desempenho**
`Import`	Inserção	Empate
`Calculate`	Seleção por faixa de valores e inserção	Cassandra
`Average`	Seleção por igualdade e inserção	MongoDB

Percebe-se que os dois sistemas têm desempenho semelhante nas operações de inserção (escrita) de dados, embora o Cassandra leve vantagem na seleção por faixa de valores e o MongoDB seja mais rápido na seleção por igualdade em um valor.

Referências

[Apaa] The Apache Software Foundation. Apache CouchDB. Disponível em `http://couchdb.apache.org/`. Acessado em 7 de agosto de 2015. 2.3.1

[Apab] The Apache Software Foundation. Eventual consistency. Disponível em `http://docs.couchdb.org/en/1.6.1/intro/consistency.html`. Acessado em 7 de agosto de 2015. 2.3.1

[Apac] The Apache Software Foundation. Replication and conflict model. Disponível em `http://docs.couchdb.org/en/1.6.1/replication/conflicts.html`. Acessado em 7 de agosto de 2015. 2.3.1

[Bas] Basho Technologies, Inc. Riak clusters. Disponível em `http://docs.basho.com/riak/latest/theory/concepts/Clusters/`. Acessado em 9 de agosto de 2015. 2.3.3

[BC14] Kelly Rosa Braghetto e Daniel Cordeiro. Introdução à modelagem e execução de workflows científicos. *Atualizações em Informática. 1ed. Porto Alegre: SBC*, páginas 1–40, 2014. Disponível em `http://www.ime.usp.br/~kellyrb/files/jai2014-cap-workflows.pdf`. Acessado em 15 de abril de 2015. 1, 4, 3, 5, 3.1

[BCD+08] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, Mei-Hui Su, e K. Vahi. Characterization of scientific workflows. Em *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*, páginas 1–10, Nov 2008. 3

[Bre00] Eric A Brewer. Towards robust distributed systems. Em *PODC*, volume 7, 2000. 2.1

[Cal] California Institute of Technology. Montage: An astronomical image mosaic engine. Disponível em `http://montage.ipac.caltech.edu/`. Acessado em 18 de outubro de 2015. 5

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, e Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008. 2

[Data] DataStax. Apache Cassandra 1.2: Partitioners. Disponível em `https://docs.datastax.com/en/cassandra/1.2/cassandra/architecture/architecturePartitionerAbout_c.html`. Acessado em 30 de novembro de 2015. 1

[Datb] DataStax. Cassandra essentials tutorials: Understanding data consistency in Cassandra. Disponível em `http://www.datastax.com/resources/tutorials/data-consistency`. Acessado em 7 de agosto de 2015. 2.3.2

[Datc] DataStax. Cassandra essentials tutorials: Understanding partitioning and replication in Cassandra. Disponível em `http://www.datastax.com/resources/tutorials/partitioning-and-replication`. Acessado em 7 de agosto de 2015. 2.3.2

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, e Werner Vogels. Dynamo: Amazon's highly available key-value store. Em *ACM SIGOPS Operating Systems Review*, volume 41, páginas 205–220. ACM, 2007. 2

[DSS+05] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005. 3.1

[FS13] Martin Fowler e Pramod J. Sadalage. *NoSQL essencial: um guia conciso para o mundo emergente da persistência poliglota*. Novatec, 1ª edição, 2013. 2, 2.1, 1, 2.1, 2.2, 2.3.1, 2.3.2, 2.3.3

[GL02] Seth Gilbert e Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, junho 2002. 2.1, 2, 3

[JDB+12] Gideon Juve, Ewa Deelman, G Bruce Berriman, Benjamin P Berman, e Philip Maechling. An evaluation of the cost and performance of scientific workflows on Amazon EC2. *Journal of Grid Computing*, 10(1):5–21, 2012. 1

[JDV+10] Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, Bruce Berriman, Benjamin P. Berman, e Phil Maechling. Data sharing options for scientific workflows on Amazon EC2. Em *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, páginas 1–9, Washington, DC, USA, 2010. IEEE Computer Society. 1

[Mona] MongoDB, Inc. Read preference. Disponível em `http://docs.mongodb.org/manual/core/read-preference/`. Acessado em 7 de agosto de 2015. 2.3.1

[Monb] MongoDB, Inc. Replication introduction. Disponível em `http://docs.mongodb.org/manual/core/replication-introduction/`. Acessado em 7 de agosto de 2015. 2.3.1, 2.3.1

[Monc] MongoDB, Inc. Sharding introduction. Disponível em `http://docs.mongodb.org/manual/core/sharding-introduction/`. Acessado em 7 de agosto de 2015. 2.3.1

[OCFdS+15] Ricardo Oda, Daniel Cordeiro, Rafael Ferreira da Silva, Ewa Deelman, e Kelly Braghetto. The case for resource sharing in scientific workflow executions. Em *XVI Simpósio em Sistemas Computacionais de Alto Desempenho*, WSCAD'15, 2015. 3.1

[Pyt] Python Software Foundation. Python. Disponível em `https://www.python.org/`. Acessado em 30 de novembro de 2015. 4.2.1

[RSZ+07] A. Ramakrishnan, G. Singh, Henan Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, e M. Samidi. Scheduling data-intensive workflows onto storage-constrained distributed resources. Em *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, páginas 401–409, maio 2007. 1

[RWH11] Charles Reiss, John Wilkes, e Joseph L. Hellerstein. Google cluster-usage traces: format + schema. Relatório técnico, Google Inc., Mountain View, CA, USA, novembro 2011. `http://code.google.com/p/googleclusterdata/wiki/TraceVersion2`. 4.1

[Wat15] Elaine Naomi Watanabe. Gerenciamento de grandes volumes de dados na execução de workflows científicos em nuvens computacionais. janeiro 2015. Qualificação de mestrado. 1

[ZKLH13] Marcelo Zuffo, Sérgio Kofuji, Roseli Lopes, e Adilson Hira. A computação em nuvem na Universidade de São Paulo. *Revista USP*, 0(97):9–18, 2013. 4.3